Moishe Halibard and Moshe Rubin

A Multiple Substring Search Algorithm

You will surely find this generalization of `find_first_of` for multiple substrings both elegant and useful. What's more, it's very efficient.

Introduction

The problem of searching for a substring within a given string is well researched, and many fast, optimized algorithms for differing needs have been published [1]. However, we found no published efficient algorithm for searching for multiple substrings within a string. An example of this would be finding the first occurrence of any of the substrings `apple`, `orange`, `pear`, or `banana` within the string `"The fruit bowl contained a large variety of fruits, including oranges, lemons and grapefruits"`.

In its simplest form, using a brute force method, one can search for each substring one at a time, looping through the string for each search. However, this is clearly not optimized (i.e., $O(N^2)$) and more efficient methods must intuitively be possible.

Kernighan [2], when writing an email spam filter, encountered exactly this problem when scanning incoming emails for a large number of characteristic spam substrings. He initially used the brute force method to scan the incoming email, but found that the spam filter was unacceptably slow. He found that the repeated scanning of the email for each substring proved to be the bottleneck in the filter code. To clear this bottleneck, he made a major improvement to the brute force method by setting up an array of the initial letter of each of the spam substrings. The array elements contained pointers to each of the substrings starting with that letter. The array typically has far fewer entries than the number of substrings, since many of the substrings share common initial letters. This reduction becomes more pronounced when there are a large number of substrings.

Kernighan then made a loop through this array into the primary loop. He searched the email message once for each entry in the initial-letter array. For each loop iteration, he progressed through the entire string, pausing to check when he encountered a character identical to the one represented by that array element. In these cases, he checked for each of the substrings starting with that array entry. His solution cleared the bottleneck in his spam filter and was therefore not generalized to a fully optimized solution, which would entirely eliminate repeatedly scanning the string.

This article summarizes the algorithm we developed to search efficiently and simultaneously for multiple substrings within a string. The algorithm uses a single pass through the string, with occasional backtracking. Possible extensions to this algorithm are discussed at the end of the article.

Outline

The program must make a single pass through a string, simultaneously searching for any of several substrings. To accomplish this task, the program must first preprocess the substrings to build an FSM (finite state machine) [3]. An FSM is a matrix of states. Each state represents a scenario within the search. In other words, the state describes a partial result of a substring search and defines what character(s) must occur next if a substring is present. Each state is assigned a numerical value, and each state points to the next state in the chain. The matrix therefore defines a path through the states that indicates the successful discovery of a substring.

Once this FSM is prepared, a single pass is made through the string. The program processes each character in the string and traverses the FSM matrix accordingly. The current state and the current character define a matrix value that points to a new state. Some small-scale backtracking is required after following false leads, but globally, this is a one-pass algorithm, $O(n \log n)$, scaling with the length of the string and the sum total length of all the substrings.

The implementation in this article returns the offset within the string of the first matching substring and stops processing.

Building the FSM

The FSM is a matrix bounded in one dimension by the size of the character alphabet, and in the other, by the maximum theoretical number of states — in our case, the sum of the lengths of all the substrings. At startup time, all the matrix entries are set to zero.

Starting with state `0` and the first character of the first substring, you set the variable at matrix position `[0][char value]` to the state to jump to if that character is encountered. This will be the first unused state. After reading the first character, this will be `1`. Setting the current state to `1` and moving to the next character in the substring, you set the variable at matrix position `[current state] [char value]` to the state to jump to if that character is found. The same procedure is followed for all the characters in the initial substring. When the final substring character is reached, the value entered into the FSM should be some special value (e.g., `-1`) to denote complete matching.

Moishe Halibard is a Windows Internals programmer at Alchemedia Technologies, Inc. in Beit Shemesh, Israel. He can be reached at `mhalibard@alchemedia.com`.

Moshe Rubin heads up the Windows Internals group at Alchemedia Technologies, Inc. in Beit Shemesh, Israel. He can be reached at `mrubin@alchemedia.com`.

For example, after entering the single substring banana, the FSM would resemble Table 1 (zeros are not shown).

Here is pseudocode for entering a single substring into the FSM:

```
for each character in the substring
{
  if this is the last character of substring
    set the matrix entry for this character
    and state to -1
  else
    set the matrix entry for this character
    and state to the next available state
    and set the current state equal to the next available
    state.
}
```

The substring banana is therefore processed as follows (see Table 1):

Table 1: *The FSM after entering the single substring banana*

	a	b	c	d	e	...	n	...	r	...	t
0		1									
1	2										
2							3				
3	4										
4							5				
5	-1										
....											

1. At initial state 0 and at the matrix position corresponding to the first character in the substring (b in this case), record the value of the next unused state (1 in this case).
2. Read the next character in the substring. At the state referenced in the previous step (1 in this case) and at the matrix position corresponding to the new character (a in this case), record the value of the next unused state (2 in this case).
3. Process subsequent characters in the substring as described in Step 2.
4. When you reach the last character in the substring, record a -1 in the matrix, signifying the end of the substring.

The next substrings are entered in an identical fashion, also starting from initial state 0. If any entries match those of earlier strings, for example entering banner after banana, their state path can be followed until the substrings diverge, whereupon a further state needs to be instantiated.

Table 1 would then resemble Table 2, with the changed entries shown in bold.

The second substring banner is processed as follows (see Table 2):

1. The first three letters of banner are identical to the first three letters of banana. Therefore, these characters are processed exactly as before, and the result is that the first three states of the FSM (states 0, 1, and 2) are unchanged.
2. The fourth character of banner is different from the fourth character of banana. In state 3 (representing character 4), record the next unused state for the value corresponding to the letter n. The next unused state in this case is state 6 (see Table 2).
3. Read the next character of banner (e). At state 6, record the next unused state (7) for the letter e (see Table 2).
4. At state 7, record a -1 for the character r, signifying the end of the substring.

It is worth noting that in this example the number of used states, seven, is far less than the theoretical maximum of *6+6=12*.

You can use this method to add additional substrings to the FSM. For example, Table 3 shows the substrings banana, banner, banned, banter, adder, red, and tab. Note that the new words banned and banter diverge from the original ban chain in a fashion similar to the word banner, as described in Table 2. The substrings adder, red, and tab have new initial characters that begin new chains from the 0 state. The value recorded for each of these characters is the next unused state at the time the substring is processed (in this case, 10 for a, 14 for r, and 16 for t — see Table 3).

Extra care is required when entering strings that are truncated or elongated versions of previous strings. In the truncated case (e.g., the substring ban, being entered after banana), the special matching value (-1) should be set at the earlier terminating position. In the

Table 2: *The FSM after adding the second substring banner*

	a	b	c	d	e	...	n	...	r	...	t
0		1									
1	2										
2							3				
3	4						**6**				
4							5				
5	-1										
6					**7**						
7									**-1**		
....											

elongated case (e.g., the substring banana being entered after ban), the special end value (-1), which denotes the end of the shorter substring, should not be changed. After reaching the -1 end value of the previous string, no more of the longer substring needs to be entered.

Table 3: *Adding the substrings banana, banner, banned, banter, adder, red, and tab*

	a	b	c	d	e	...	n	...	r	...	t
0	10	1							14		16
1	2										
2							3				
3	4						6				8
4							5				
5	-1										
6					7						
7				-1					-1		
8					9						
9									-1		
10				11							
11				12							
12					13						
13									-1		
14					15						
15				-1							
16	17										
17		-1									
....											

This implementation precludes finding any elongated substrings, if truncated versions are also present, which is fine if you are interested in finding only the first of any substring matches. If all substring matches are needed, instead of denoting termination with a simple -1, the state values would have to set aside one bit to denote termination, while allowing the other bits to contain a value denoting the state path of any continuing substring.

As an example of entering a truncated version of a previous string, if the next string entered were ban, the only change to the FSM would be inserting -1 at the end of the shorter string. The table now looks like Table 4.

Notice how entry [2][n] has changed from 3 to -1. If the string contains any of the longer strings that start with ban (i.e., banana, banner, banned, and banter), you will now always first find ban, which is in fact the desired result.

Sample code to enter a single substring into the FSM is shown in Listing 1 (BuildingFSM.cpp). Note that this code expects all the substring characters to have positive ASCII values (i.e., no greater than 0x7F). In order for all extended ASCII character values to occur in the substrings (i.e., up to 0xFF), they must be cast to unsigned char before they are entered into the FSM.

Searching for All Substrings Using the FSM

Once the FSM is prepared, the program must search the string for any occurrences of the substrings. This search consists of reading each character in the string and proceeding accordingly through the FSM matrix. As each character is read, the program checks the FSM for the value associated with the current state and the current character. If that value is zero, no substring starts with the current letter, and you progress to the next character of the string. If, however, the value is non-zero, you have possibly found a matching substring, so you store the current string position (to return to in the case of a false lead). The value obtained from the matrix becomes the new state. You continue the character-by-character search, until you either reach the special value (-1) denoting a complete match, or you return to state 0. In the former case, a match has been found.

Table 4: *Entering a truncated version of a previous string*

	a	b	c	d	e	...	n	...	r	...	t
0	10	1							14		16
1	2										
2							-1				
3	4						6				8
4							5				
5	-1										
6					7						
7				-1					-1		
8					9						
9									-1		
10				11							
11				12							
12					13						
13									-1		
14					15						
15				-1							
16	17										
17		-1									
....											

In the latter case, the search must backtrack and continue from where it left the 0 state, which you have saved.

For example, consider the FSM populated as shown in Figure 1, for the strings banana, banner, banter, banned, ban, adder, red, and tab. Encountering the character b in initial state 0, you would see from the FSM that the current state should be set to 1. If an a followed, you would set the current state to 2. If an n now followed, the state would be set to -1, reporting a complete match. If instead any other letter followed the a, the state would be reset to 0, and searching would resume from the next character after the start of the false trail — the character a. Since the FSM reports that the value for this state (0) and character (a) is 10, the start of another substring, the state is set to 10 and searching simply continues onwards. If any letter other than d follows, the state is once again reset to 0, and the value for that letter and state 0 is looked up.

Although the FSM is an algorithmically compact representation of the saved substrings, it is difficult to visualize its contents and the searching method using it. It is more helpful to visualize it using Figure 1, which is in fact algorithmically identical. Using Figure 1, you can follow the path of any substring and set the current state based on the next character. Any character found that is not connected with an arrow from the given state reverts the state back to the starting state of 0.

Listing 1: *BuildingFSM.cpp* — ***Sample code to enter a single substring into the FSM***

```
// Enter each substring (pSubStr), in turn, stopping when we
// reach its end (lSubstringLength), or if we reach a previous
// full-string match (denoted by lCurrentState==-1)
// Only create new states when they are needed, and then increment
// the lMaxUsedState counter

long lCurrentState = 0;
for (int i=0; (i<lSubstringLength)  &&  (lCurrentState!=-1); i++)
{
  // special case - last character of substring
  if (i == (lSubstringLength-1) )
    lCurrentState = FSM[lCurrentState][pSubStr[i]] = -1;
  else
  {
    // add to FSM, if current entry for this character value
    //  and state is still zero
    if (FSM[lCurrentState][pSubStr[i]] == 0)
      lCurrentState = FSM[lCurrentState][pSubStr[i]] =
        ++lMaxUsedState;
    // or simply record an existing value as the current state
    // (following state path of a previous substring)
    else
      lCurrentState = FSM[lCurrentState][pSubStr[i]];
  }
}
```

— End of Listing —

Pseudocode for Searching

```
for each character in the string
{
  Save the value of the current state
  (called the previous state)

  If the current state is 0, save a pointer to
  this character (called starting character),
  since this may be the start of a full match.

  The value in the FSM for the current state and
  character becomes the new current state

  If the new current state is -1, a full match
  has been found, so return the pointer to the
  starting character
```

Listing 2: *SearchingFSM.cpp* — ***Search string*** *pStr* ***of length*** *strlength*

```
// start in state 0
long lCurrentState = 0;
long iStartIndex = 0;

for(int i=0; i<strlength; i++)
{
  long lPrevState = lCurrentState;

  // save a pointer to the starting character,
  // if we are in state  0
  if(lCurrentState==0)
    iStartIndex=i;

  // follow up the table states
  lCurrentState = FSM[lCurrentState][(unsigned char)pStr[i]];

  // if we reach a -1, return the position of the
  // start of the substring - we have found a full match
  if(lCurrentState == -1)
    return iStartIndex;

  // have we followed a false lead?
  // if so, backtrack i to position to continue search
  // with state reset to zero set i to iStartIndex,
  // and it will be incremented as the loop re-enters
  if(lPrevState !=  0 && lCurrentState == 0)
    i = iStartIndex;
}

// If we get here, no matches were found
return -1;
```

— End of Listing —

```
  If the new state is 0, while the previous state
  was non-zero, we have followed a false lead
  So, backtrack the current search position to the
  character following the starting character,
  and reset the current state to 0

  Progress to the next character in the string
}
```

If the loop exits with no matches found, no substrings exist within the string.

Sample code to search a string (`pStr`) of length `strlength` is shown in Listing 2 (`SearchingFSM.cpp`). As with Listing 1, this code expects all the substring characters to have positive ASCII values.

The backtracking done after a false lead has been followed is an obvious inefficiency. However, the speed for several substring searches was so fast with this algorithm that no need was found to optimize it further.

Extensions

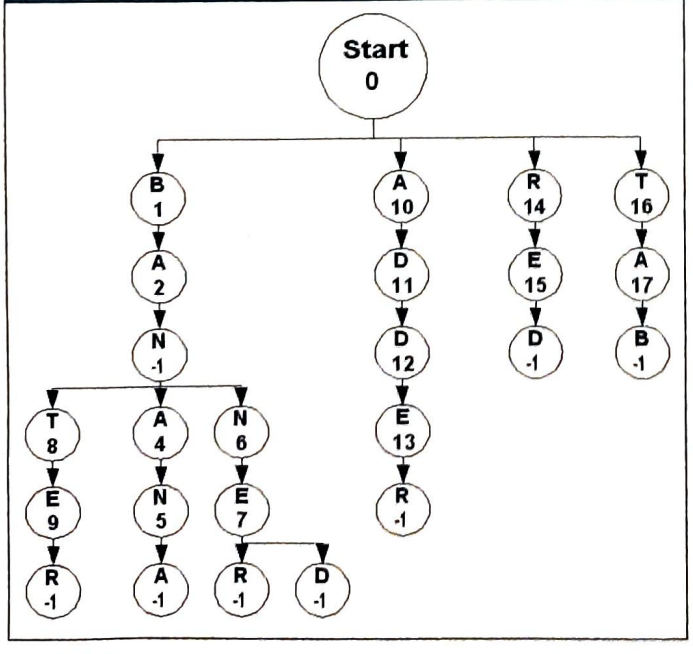
Some simple amendments can be made to the algorithm presented here.

If the identity of the found substring is required, the FSM could use a negative index to signify the end of a substring, rather than a simple `-1` value. The index could simply be the count of substrings entered so that the first one entered would have value `-1`, the second `-2`, and so on. In this manner, a negative value encountered in the FSM would signify both termination of a substring and the identity of the substring found.

If all substring matches are required, rather than using a `-1` to denote string termination, a high-order bit could be set denoting termination, with the rest of the integer value storing the next state to move to if longer strings exist.

The backtracking done after a false lead has been followed could be further optimized. Such optimization lies at the core of the Boyer-Moore algorithm [1]. As an example, consider if the string contains the characters `ADDED`. Using the FSM described above, the algorithm fails on the final `D` and backtracks to the first `D`. However, there is no substring starting with `D` or `E`, so no backtracking should be done at all. Instead the character following `ADDED` should be investigated. In order to implement this "clever backtracking," you would need to post-process the FSM, adding to each non-zero entry a value denoting how many characters to skip when backtracking. This would require the FSM to be a matrix of structures rather than simple integers and the development of an algorithm for post-processing the FSM.

Figure 1: *The FSM populated for the strings banana, banner, banter, banned, ban, adder, red, and tab*



Conclusion

The FSM is a powerful tool and is ideally suited to searching for multiple substrings within a string. The algorithm presented is robust and efficient and joins the growing number of published string searching algorithms, solving a problem not previously addressed in the open literature. ❑

References

[1] An excellent summary of many string searching algorithms, as well as code to implement them, can be found at <www.dir.univ-rouen.fr/~charras/string/string.html>.

[2] B.W.Kernighan and B. Pike. *The Practice of Programming* (Addison-Wesley, 1999).

[3] FSMs are discussed in numerous texts on algorithms. For example, see R. Sedgewick. *Algorithms in C* (Addison-Wesley, 2001).